Chapter 3

Anatomy of C

In This Chapter

- Reviewing parts of the C language
- Understanding keywords and functions
- Exploring operators, variables, and values
- Learning to comment
- Building a basic C language skeleton
- Printing a math problem

All programming languages consist of instructions that tell a computer or another electronic device what to do. Though basic programming concepts remain the same, each language is different, created to fulfill a specific need or to frustrate a new crop of college freshmen. The C language meets both of those qualifications, being flexible and intimidating. To begin a relationship with C in a friendly, positive way, get to know the language and how it works.

You'll probably want to reread this chapter after you venture deep into Part II of this book.

Parts of the C Language

Unlike a human language, C has no declensions or cases. You'll find no masculine, feminine, or neuter. And you never need to know what the words *pluperfect* and *subjunctive* mean. You do have to understand some of the lingo, the syntax, and other mischief. This section provides an overview of what's what in the C language.

Programming language levels

It's almost a tradition. Over time, hundreds of programming languages have been developed. Many fade away, yet new ones pop up on an annual basis. The variety is explained by different languages meeting specific needs.

Generally speaking, programming languages exist on three levels: low, medium, and high.

High-level languages are the easiest to read, using words and phrases found in human languages (mostly English). These languages are quick to learn, but are often limited in their flexibility.

Low-level languages are the most cryptic, often containing few, if any, recognizable human language words. These languages access hardware directly and therefore are extremely fast. The drawback is that development time is slow because pretty much everything has to be done from scratch.

Midlevel languages combine aspects from both high- and low-level languages. As such, these languages are quite versatile, and the programs can be designed to do just about anything. C is the prime example of a midlevel programming language.

Keywords

Forget nouns, verbs, adjectives, and adverbs. The C language has *keywords.* Unlike human languages, where you need to know at least 2,000 words or so to be somewhat literate, the C language sports a scant vocabulary: Only a handful of keywords exist, and you may never use them all. Table 3-1 lists the 44 keywords of the C language.

Table 3-1 C Language Keywords

_Alignas	break	float	signed
_Alignof	case	for	sizeof
_Atomic	char	goto	static
_Bool	const	if	struct
_Complex	continue	inline	switch
_Generic	default	int	typedef
_Imaginary	do	long	union
_Noreturn	double	register	unsigned
_Static_assert	else	restrict	void
_Thread_local	enum	return	volatile
auto	extern	short	while

The keywords shown in Table 3-1 represent the C language's basic commands. These simple directions are combined in various interesting ways to do wondrous things. But the language doesn't stop at keywords; continue reading in the next section.

- Don't bother memorizing the list of keywords. Though I still know the 23 "to be" words in English (and in the same order as specified in my eighth grade English text), I've never memorized the C language keywords.
- The keywords are all case-sensitive, as shown in Table 3-1.
- Of the 44 keywords, 32 are original C language keywords. The C99 update (in 1999) added five more, and the more recent C11 (2011) update added seven. Most of the newer keywords begin with an underscore, as in `_Alignas`.
- Keywords are also known as *reserved words,* which means that you cannot name functions or variables the same as keywords. The compiler moans like a drunken, partisan political blogger when you attempt to do so.

Functions

Where you find only 44 keywords, there are hundreds (if not thousands) of functions in the C language, including functions you create. Think of a function as a programming machine that accomplishes a task. Truly, functions are the workhorses of the C language.

The telltale sign of the function is the appearance of parentheses, as in `puts()` for the `puts` function, which displays text. Specifically, *puts* means "put string," where *string* is the programming lingo for text that's longer than a single character.

Functions are used in several ways. For example, a `beep()` function may cause a computer's speaker to beep:

```
beep();
```

Some functions are sent values, as in

```
puts("Greetings, human.");
```

Here, the string `Greetings, human.` (including the period) is sent to the `puts()` function, to be sent to standard output or displayed on the screen. The double quotes define the string; they aren't sent to standard output. The information in the parentheses is said to be the function's *arguments,* or *values.* They are *passed* to the function.

Functions can *generate,* or return, information as well:

```
value = random();
```

The `random()` function generates a random number, which is returned from the function and stored in the variable named `value`. Functions in C return only one value at a time. They can also return nothing. The function's documentation explains what the function returns.

Functions can also be sent information or return something:

Functions can also be sent information as well as return something:

```
result = sqrt(256);
```

The `sqrt()` function is sent the value 256. It then calculates the square root of that value. The result is calculated and returned, stored in the `result` variable.

- See the later section "Variables and values" for a discussion of what a variable is.
- A function in C must be defined before it's used. That definition is called a *prototype.* It's necessary so that the compiler understands how your code is using the function.
- You'll find lists of all the C language functions online, in what are called *C library references.*
- Function prototypes are held in *header files,* which must be included in your source code. See the later section "Adding a function."
- The functions themselves are stored in C language libraries. A *library* is a collection of functions and the code that executes those functions. When you link your program, the linker incorporates the functions' code into the final program.
- As with keywords, functions are case sensitive.

Operators

Mixed in with functions and keywords are various symbols collectively known as *operators.* Most of them are mathematic in origin, including traditional symbols like the plus (+), minus (–), and equal (=) signs.

Operators get thrown in with functions, keywords, and other parts of the C language; for example:

```
result = 5 + sqrt(value);
```

Here, the = and + operators are used to concoct some sort of mathematical mumbo jumbo.

Not all C language operators perform math. Refer to Appendix C for the lot.

Variables and values

A program works by manipulating information stored in variables. A *variable* is a container into which you can stuff values, characters, or other forms of information. The program can also work on specific, unchanging values that I call *immediate* values:

```
result = 5 + sqrt(value);
```

In this example, `result` and `value` are variables; their content is unknown by looking at the code, and the content can change as the program runs. The number `5` is an immediate value.

C sports different types of variables, each designed to hold specific values. Chapter 6 explains variables and values in more detail.

Statements and structure

As with human languages, programming languages feature *syntax* — it's the method by which the pieces fit together. Unlike English, where syntax can be determined by rolling dice, the method by which C puts together keywords, functions, operators, variables, and values is quite strict.

The core of the C language is the *statement,* which is similar to a sentence in English. A statement is an action, a direction that the program gives to the hardware. All C language statements end with a semicolon, the programming equivalent of a period:

```
beep();
```

Here, the single function `beep()` is a statement. It can be that simple. In fact, a single semicolon on a line can be a statement:

```
;
```

The preceding statement does nothing.

Statements in C are executed one after the other, beginning at the top of the source code and working down to the bottom. Ways exist to change that order as the program runs, which you'll see elsewhere in this book.

The paragraph-level syntax for the C language involves the use of curly brackets, or *braces*. They enclose several statements as a group:

```
{
    if( money < 0 ) getjob();
    party();
    sleep(24);
}
```

These three statements are held within curly brackets, indicating that they belong together. They're either part of a function or part of a loop or something similar. Regardless, they all go together and are executed one after the other.

You'll notice that the statements held within the curly brackets are indented one tab stop. That's a tradition in C, but it's not required. The term *white space* is used to refer to tabs, empty lines, and other blank parts of your source code.

Generally, the C compiler ignores white space, looking instead for semicolons and curly brackets. For example, you can edit the source code from project ex0201 to read:

```
#include <stdio.h>
int main(){puts("Greetings, human.");return 0;}
```

That's two lines of source code where before you saw several. The `#include` directive must be on a line by itself, but the C code can be all scrunched up with no white space. The code still runs.

Thankfully, most programmers use white space to make their code more readable.

- The most common mistake made by beginning C language programmers is forgetting to place the semicolon after a statement. It may also be the most common mistake made by experienced programmers!
- The compiler is the tool that finds missing semicolons. That's because when you forget the semicolon, the compiler assumes that two statements are really one statement. The effect is that the compiler becomes confused and, therefore, in a fit of panic, flags those lines of source code as an error.

Comments

Some items in your C language source code are parts of neither the language nor the structure. Those are comments, which can be information about the program, notes to yourself, or filthy limericks.

Traditional C comments begin with the /* characters and end with the */ characters. All text between these two markers is ignored by the compiler, shunned by the linker, and avoided in the final program.

Listing 3-1 shows an update to the code from project ex0201 where comments have been liberally added.

Listing 3-1: Overly Commented Source Code

```
/* Author: Dan Gookin */
/* This program displays text on the screen */

#include <stdio.h>    /* Required for puts() */

int main()
{
    puts("Greetings, human.");  /* Displays text */
    return 0;
}
```

You can see comments in Listing 3-1. A comment can appear on a line by itself or at the end of a line.

The first two lines can be combined for a multiline comment, as shown in Listing 3-2.

Listing 3-2: Multiline Comments

```
/* Author: Dan Gookin
   This program displays text on the screen */

#include <stdio.h>    /* Required for puts() */

int main()
{
    puts("Greetings, human.");  /* Displays text */
    return 0;
}
```

All text between the /* and the */ is ignored. The Code::Blocks editor displays commented text in a unique color, which further confirms how the compiler sees and ignores the comment text. Go ahead and edit the ex0201 source code to see how comments work.

A second comment style uses the double-slash (//) characters. This type of comment affects text on one line, from the // characters to the end of the line, as shown in Listing 3-3.

Listing 3-3: Double-Slash Comments

```
#include <stdio.h>

int main()
{
    puts("Greetings, human.");  // Displays text
    return 0;
}
```

Don't worry about putting comments in your text at this point, unless you're at a university somewhere and the professor is being anal about it. Comments are for you, the programmer, to help you understand your code and remember what your intentions are. They come in handy down the road, when you're looking at your code and not fully understanding what you were doing. That happens frequently.

Behold the Typical C Program

All C programs feature a basic structure, which is easily shown by looking at the C source code skeleton that Code::Blocks uses to start a new project, as shown in Listing 3-4.

Listing 3-4: Code::Blocks C Skeleton

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

This listing isn't the bare minimum, but it gives a rough idea of the basic C program.

- Just as you read text on a page, C source code flows from the top down. The program starts execution at the first line, and then the next line, and so on until the end of the source code. Exceptions to this order include decision-making structures and loops, but mostly the code runs from the top down.
- Decision-making structures are covered in Chapter 8; loops are introduced in Chapter 9.

Understanding C program structure

To better understand how C programs come into being, you can create the simplest, most useless type of C program.

Exercise 3-1: Follow the steps in this section to create a new Code::Blocks project, ex0301.

Here are the specific steps:

1. **Start a new Code::Blocks project:** ex0301**.**
2. **Edit the source code to match Listing 3-5.**

Listing 3-5: A Simple Program That Does Nothing

```
```

That's not a misprint. The source code for `main.c` is blank, empty. Simply erase the skeleton that Code::Blocks provided.

3. **Save the project.**
4. **Build and run.**

   Code::Blocks complains that the project hasn't been built yet. Tough!

5. **Click the Yes button to proceed with building the project.**

   Nothing happens.

Because the source code is empty, no object code is generated. Further, the program that's created (if a program was created) is empty. It does nothing. That's what you told the compiler to do, and the resulting program did it well.

You may see a Code::Blocks error message after Step 4. That's because the IDE was directing the operating system to run a program in a command prompt window. The error you see is the reference to a program file that either doesn't exist or doesn't do anything.

Setting the main() function

All C programs have a `main()` function. It's the first function that's run when a program starts. As a function, it requires parentheses but also curly brackets to hold the function's statements, as shown in Listing 3-6.

Continue with Exercise 3-1: Rebuild the source code for project ex0301, as shown in Listing 3-6. Save the project. Build and run.

Listing 3-6: The main() Function

```
main() {}
```

This time, you see the command prompt window, but nothing is output. That's great! You didn't direct the code to do anything, and it did it well. What you see is the minimum C program. It's also known as the *dummy* program.

- `main` isn't a keyword; it's a function. It's the required first function in all C language source code.
- Unlike other functions, `main()` doesn't need to be declared. It does, however, use specific arguments, which is a topic covered in Chapter 15.

Returning something to the operating system

Proper protocol requires that when a program quits, it provides a value to the operating system. Call it a sign of respect. That value is an integer (a whole number), usually zero, but sometimes other values are used, depending on what the program does and what the operating system expects.

Continue with Exercise 3-1: Update the source code for project ex0301 to reflect the changes shown in Listing 3-7.

Listing 3-7: Adding the Return Statement

```
int main()
{
    return(1);
}
```

First, you're declaring the `main()` function to be an integer function. The `int` tells the compiler that `main()` returns, or *generates,* an integer value.

The `return` statement passes the value `1` back to the operating system, effectively ending the `main()` function and, therefore, the program.

As you type `return`, Code::Blocks may display Auto Complete text, as shown in Figure 3-1. These hints are useful to help you code, though at this point in your programming career, you can freely ignore them.

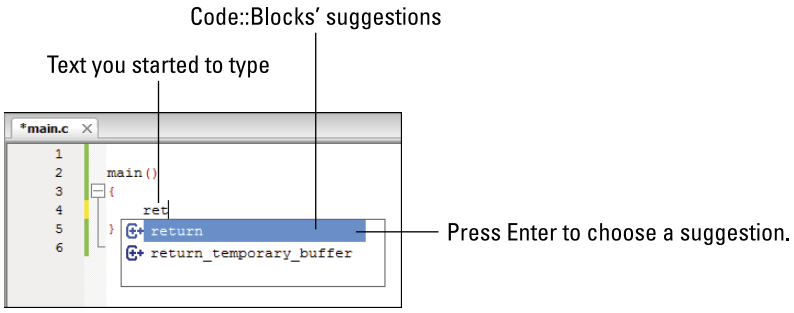

Figure 3-1: Code::Blocks' Auto Complete hints.

Continue with Exercise 3-1: Save, build, and run the project.

The results are similar to the previous run, but you'll notice the return value of `1` specified in the Code::Blocks summary in the command prompt window:

```
Process returned 1 (0x1)
```

If you like, edit the code again and change the return value to something else — say, 5. That value appears in the Code::Blocks output when you run the project.

- Traditionally, a return value of `0` is used to indicate that a program has completed its job successfully.
- Return values of `1` or greater often indicate some type of error, or perhaps they indicate the results of an operation.
- The keyword `return` can be used in a statement with or without parentheses. Here it is without them:

  ```
  return 1;
  ```

  In Listing 3-7, `return` is used with parentheses. The result is the same. I prefer to code `return` with parentheses, which is how it's shown throughout this book.

Adding a function

C programs should do something. Though you can use keywords and operators to have a program do marvelous things, the way to make those things useful is output. Continue working on this chapter's example:

Continue with Exercise 3-1: Modify the project's source code one final time to match Listing 3-8.

Listing 3-8: More Updates for the Project

```
#include <stdio.h>

int main()
{
    printf("4 times 5 is %d\n",4*5);
    return(0);
}
```

You're adding three lines. First, add the `#include` line, which brings in the `printf()` function's prototype. Second, type a blank line to separate the processor directive from the `main()` function. Third, add the line with the `printf()` function. All functions must be declared before use, and the `stdio.h` file contains the declaration for `printf()`.

When you type the first " for `printf()`, you see the second quote appear automatically. Again, that's Code::Blocks helping you out. Remain calm and refer to Figure 3-2 for information on other things you may see on the screen.

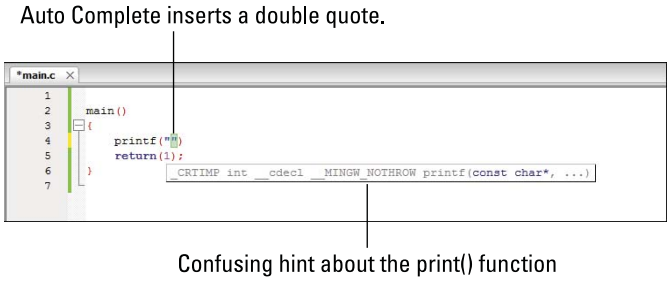

Figure 3-2: Typing a function in Code::Blocks.

Before proceeding, please note these two important items in your source code:

- Ensure that you typed the `#include` line exactly as written:

  ```
  #include <stdio.h>
  ```

  The `#include` directive tells the compiler to fetch the header file, `stdio.h`. The header file is required in order to use the `printf()` function.

- Ensure that you type the `printf()` statement exactly as written:

  The `printf()` function sends formatted text to the standard output device: the display. It also contains a math problem, `4*5`. The result of that problem is calculated by the computer and then displayed in the formatted text:

```
printf("4 times 5 is %d\n",4*5);
```

You'll find lots of important items in the `printf()` statement, each of which is required: quotes, comma, and semicolon. Don't forget anything!

Later chapters cover the `printf()` function in more detail, so don't worry if you're not taking it all in at this point.

Finally, I've changed the return value from 1 to 0, the traditional value that's passed back to the operating system.

Continue with Exercise 3-1: Save the project's source code. Build and run.

If you get an error, double-check the source code. Otherwise, the result appears in the terminal window, looking something like this:

```
4 times 5 is 20
```

The basic C program is what you've seen presented in this section, as built upon over the past several sections. The functions you use will change, and you'll learn how things work and become more comfortable as you explore the C language.

Where are the files?

A C programming project needs more than just the source code: It includes header files and libraries. The header files are called in by using the `#include` statement; libraries are brought in by the linker. Don't worry about these files, because the IDE — specifically, the compiler and linker — handle the details for you.

Because C comes from a Unix background, traditional locations for the header and library files are used. Header files are found in the `/usr/include` directory (folder). The library files dwell in the `/usr/lib` directory. Those are system folders, so look but *don't touch* the contents. I frequently peruse header files to look for hints or information that may not be obvious from the C language documentation. (Header files are plain text; library files are data.)

If you're using a Unix-like operating system, you can visit those directories and peruse the multitude of files located there. On a Windows system, the files are kept with the compiler; usually, in `include` and `lib` folders relative to the compiler's location.